Lazy evaluation, unendliche Datenstrukturen

```
primes :: [ Int ]
primes = sieve [ 2 .. ]

sieve :: [ Int ] -> [ Int ]
sieve (p : qs) =
    p : sieve ( filter ( \ q -> 0 /= q `rem` p ) qs )
```


Syntax

```
module Direction where
import Monad (guard)
data Direction = Direction { direction :: (Int, Int) }
     deriving (Eq, Ord, Show, Read)
directions :: [ Direction ]
directions = do
    dx <- [ -1 .. 1 ]
    dy <- [ -1 .. 1 ]
    guard $ not $ 0 == dx && 0 == dy
    return $ Direction ( dx, dy )
```

beachte implizite `{ ; }` durch Layout-Regel

Praktische Haskell-Programmierung

Johannes Waldmann, Institut für Informatik, Uni Leipzig

- `http://www.haskell.org`:
  - Definition von Sprache und Bibliotheken
  - Links zu Implementierungen (`hugs`, `ghc`, `nhc`)
- Haskell-Kompakt-Kurs (Code-Beispiele): `http://www.informatik.uni-leipzig.de/~joe/edu/ws01/hc/`
- lokale Mail-Liste (+ Archiv): `http://theopc.informatik.uni-leipzig.de/mailman/listinfo/Haskell`

Funktoren

```
class Functor f where fmap :: (a -> b) -> (f a -> f b)

instance Functor [] where
   fmap f [] = [] ; fmap f (x : xs) = f x : fmap f xs

data Tree a = Node { key :: a, children :: [ Tree a ] }
instance Functor Tree where
    fmap f n = Node { key = f (key n)
                    , children = fmap (fmap f) (children n)
                    }
instance Functor IO where
    fmap f action = do x <- action; return $ f x
```


Haskell-Typklassen ∼ Java-Interfaces

```
class Show a where show :: a -> String

instance Show Bool where
    show True = "True"; show False = "False"

instance (Show a, Show b) => Show (a, b) where
    show (x,y) = "(" ++ show x ++ "," ++ show y ++ ")"

class Eq a where (==) :: a -> a -> Bool
class Eq a => Ord a where (<) :: a -> a -> Bool
```


IO-Aktionen

```
module Main (main) where
import IO ; import System

main :: IO ()
main = do argv <- getArgs ; cats argv

cats :: [ String ] -> IO ()
cats [] = return ()
cats (file : files) =
    do cs <- readFile file ; putStr cs ; cats files
```

Haskell = the world's finest imperative programming language

ghc-Bibliotheken

- `data`: balancierte Bäume (`FiniteMap`, `Set`)
- `text`: Pretty-Printer, Parser (`parsec`)
- `net`: Sockets
- `concurrent`: lightweight threads
- `posix`, `util`: `GetOpt`
- `http://haskell.org/HOpenGL/`

Standard-Bibliothek: Bsp. Directory

```
create :: [ String ] -> IO ()
create dirs = sequence_ $ do
    prefix <- inits dirs
    guard $ not $ null prefix
    let path = concat $ intersperse "/" prefix
    return $ do ok <- doesDirectoryExist path
                when ( not ok ) $ do
                     createDirectory path
```


Monaden

```
class Functor f => Monad f where
    return :: a -> f a
    bind   :: a -> (a -> f b) -> f b

instance Monad [] where
    return x = [ x ]
    bind l f = [ y | x <- l , y <- f x ]

instance Monad IO where ...
```

do-Notation

```
do x <- a            a `bind` $ \ x ->
   y <- b      ⇒       b `bind` $ \ y ->
   return z                return z
```

Beispiel: Parser-Kombinatoren (Parsec)

```
catenation :: Parser Exp
catenation = do ps <- many1 monomial; return $ foldr1 Dot ps

monomial :: Parser Exp
monomial = do
    x <- atom
    fs <- many $
            do symbol "^"
               (     do symbol  "+" ; return $ Plus
                 <|> do symbol  "*" ; return $ Star
                 <|> do e <- natural; return $ Power e
                 <?> "exponent" )
    return $ foldl (.) id (reverse fs) $ x

atom :: Parser Exp
atom =      parens expression
       <|>  do b <- basic ; whiteSpace; return b
       <?>  "atomic expression"
```


Beispiel: Layout-Kombinatoren (Pretty)

```
instance (ToDoc s, ToDoc (Set s)) => ToDoc (NFA s) where
    toDoc a =
         let pstates = text "states" <+> equals <+> toDoc (states a)
             pstarts = text "starts" <+> equals <+> toDoc (starts a)
             pfinals = text "finals" <+> equals <+> toDoc (finals a)
             ptrans  = text "trans " <+> equals <+> toDoc (trans  a)

         in      text "NFA"
             <+> braces ( fsep $ punctuate comma
                                   [ pstates, pstarts, pfinals, ptrans] )
```


Beispiel: FiniteMap, Set

```
punkt :: Ord s => s -> NFA s -> NFA s
-- addiere neuen Startzustand,
-- in den keine Pfeile hineinführen
punkt t a =
    a { states = Set.union (states a) (Set.unitSet t)
      , starts = Set.unitSet t
      , finals = if not $ Set.isEmptySet $ Set.intersect (starts a) (finals a)
                 then Set.union (finals a) (Set.unitSet t)
                 else            finals a
      , trans = addListToFM_C Set.union (trans a)
              $ do ((p, c), qs) <- ltrans a
                   guard $ Set.elementOf p (starts a)
                   return ((t, c), qs)
      }
```

HOpenGL

```
display :: IORef Int -> IORef Int -> DisplayAction
display day year = do
   clear [ColorBufferBit]
   color (Color3 1.0 1.0 1.0 :: Color3 GLfloat)
   pushMatrix
   wireSphere 1.0 20 16   -- draw sun
   y <- readIORef year
   rotate (fromIntegral y :: GLfloat) (Vector3 0 1 0)
   translate (Vector3 2 0 0 :: Vector3 GLfloat)
   d <- readIORef day
   rotate (fromIntegral d :: GLfloat) (Vector3 0 1 0)
   wireSphere 0.2 10 8    -- draw smaller planet
   popMatrix
   swapBuffers
```


Server: Sockets, Concurrent, Posix

```
server :: Int -> IO ()
server portnum = do
    Posix.installHandler Posix.sigPIPE Posix.Ignore Nothing
    let port = S.PortNumber $ fromIntegral portnum
    s <- S.listenOn port
    receptions $ Par { socket = s, components = picture }

receptions :: Par -> IO ()
receptions p = sequence_ $ repeat $ reception p

reception :: Par -> IO ()
reception p = do
    (h, rhost, rport) <- S.accept (socket p)
    Concurrent.forkIO $ handler $ p { handle = h }
```


Sockets (Client)

```
main = do
     h <- dial "theopc" "9999"
     sequence_ $ repeat $ chat h

dial :: String -> String -> IO Handle
dial ho po  =
    connectTo ho ( PortNumber $ fromIntegral $ read po )

chat h = do
     msg <- hGetLine h
     putStrLn $ "Server: " ++ msg
     user <- hGetLine stdin
     hPutStrLn h user
```